Chapter 2

Background

This chapter provides background information on garbage collection basics, reference counting, Immix garbage collection, and conservative garbage collection to place the research contributions in context.

This chapter starts with a brief introduction to the field of garbage collection in Section 2.1. Section 2.2 outlines the garbage collection terminology, Section 2.3 outlines the key components of garbage collection, and Section 2.4 describes the canonical garbage collectors. Section 2.5 provides a detailed background on reference counting with different optimizations over the last 50 years or so. Section 2.6 provides an overview of the Immix collector. Section 2.7 provides background on conservative garbage collection.

2.1 Overview of Garbage Collection

Garbage collection is an integral part of modern programming languages. It frees the programmer from manually dealing with memory deallocation for every object they create. Garbage collection was introduced in LISP [McCarthy, 1960] and it has gained popularity through Java and .NET. It is also included in languages such as Haskell, JavaScript, PHP, Perl, Python, and Smalltalk. For a more complete discussion of the fundamentals of garbage collection see [Jones et al., 2011; Wilson, 1992].

Programs require data to execute and this data is typically stored in memory. Memory can be allocated: a) statically where memory requirements for the data are fixed ahead-of-time, b) on the stack where the lifetime of the data is tightly bound with the currently executing method, and c) dynamically, where memory requirements are determined during execution – potentially changing between individual executions of the same program. Dynamically allocated memory can be managed either explicitly or automatically by the program. Popular programming languages, such as C/C++ require the programmer to explicitly manage memory through primitives such as the C function *malloc* and *free*, which is tedious and error-prone. Managed languages, such as Java/.NET use a garbage collector to automatically free memory.

The purpose of garbage collection is to reclaim memory that is no longer in use by the program. Determining precisely when an object will no longer be accessed is difficult in general, so garbage collectors usually rely on reachability to conservatively approximate liveness. An object is reachable if it is transitively reachable from the running program state. Objects that are not reachable are garbage. A garbage collector differentiates between objects in the program that are no longer reachable and thus the program is guaranteed never to access (garbage/dead) and objects that are reachable (non-garbage/live). The collector then frees the memory that garbage objects are occupying.

2.2 Terminology

The area of memory used for dynamic object allocation is known as the **heap**. The process of reclaiming unused memory is known as **garbage collection**, a term coined by McCarthy [1960]. Following Dijkstra et al. [1976], from the point of view of the garbage collector, the term **mutator** refers the application or program that mutates the heap. Collectors that must stop the mutator to perform collection work are known as **stop the world** collectors, as compared to **concurrent** or **on-the-fly** collectors which reclaim objects while the application continues to execute. Collectors that employ more than one thread to do the collection work are **parallel** collectors. A parallel collector can either be stop the world or concurrent. The term **mutator time** is used to denote the time when the mutator is running and the term **GC time** is used to denote the time when the garbage collector is running. A garbage collector that checks the liveness of all objects in the heap at each collection is known as a **full heap** collector, as compared to a **generational** or **incremental** collector which may collect only part of the heap. Some garbage collectors require interaction with the running mutator. These interactions are generally implemented with **barriers**. A barrier is inserted by the compiler on every read or write to track reference read or mutation. The most common form of barrier is a **write barrier**, which is invoked whenever the mutator writes to a reference in the heap. Some collectors require knowledge of the **runtime roots**, all references into the heap held by runtime including stacks, registers, statics, and JNI.

2.3 Garbage Collection Algorithms

Automatic memory management consists of three key components: a) object allocation, b) garbage identification, and c) garbage reclamation. Different garbage collection algorithms employ different approaches to handle each of the components.

2.3.1 Object Allocation

The allocator plays a key role in mutator performance since it determines the placement and thus locality of objects. There are two techniques used for object allocation — contiguous allocation and free-list allocation.

Contiguous Allocation Contiguous memory allocation appends new objects by incrementing a pointer by the size of the new object [Cheney, 1970]. The allocator places objects contiguously in memory in allocation order. Such allocators are simple and fast at allocation time and provide excellent locality to the mutator because objects allocated together in time are generally used together [Blackburn et al., 2004a]. Allocating them contiguously thus provides spatial cache line and page reuse. Contiguous allocation generally performs synchronized allocation of larger chunks [Garthwaite and White, 1998; Alpern et al., 1999; Berger et al., 2000], which are then assigned to a single allocation thread that performs fast and unsynchronized contiguous allocation within the chunk.

Free-list Allocation Free-list allocation divides memory into cells of various fixed sizes [Wilson et al., 1995], known as a free list. Each free list is unique to a size and is composed from blocks of contiguous memory. Free-list allocation allocates an object into a free cell in the smallest size class that accommodates the object. So it places objects in memory based on their size and free memory availability rather than allocation order. When an object becomes free, the allocator returns the cell containing the object to the free list for reuse. Free-list allocation suffers two notable shortcomings. First, it is vulnerable to fragmentation of two kinds. It suffers from internal fragmentation when objects are not perfectly matched to the size of their containing cell, and it suffers external fragmentation when free cells of particular sizes exist, but the allocator requires cells of another size. Second, it suffers from poor locality because it often positions contemporaneously allocated objects in spatially disjoint memory [Blackburn et al., 2004a].

2.3.2 Garbage Identification

There are two techniques for identifying garbage — reference counting and tracing. All garbage collection algorithms in the literature use one of these techniques for garbage identification.

Reference Counting Reference counting directly identifies garbage. It keeps track for each object a count of the number of incoming references to it held by other objects, known as a *reference count*. When a reference count falls to zero, the associated object is garbage. It is incomplete because it cannot detect cycle of garbage.

Tracing Tracing indirectly identifies garbage by directly identifying all live objects. It performs a transitive closure over the object graph, starting with the *roots* – references in the stacks, registers, statics and JNI. It identifies each visited object as live. All objects that were not visited during the trace are identified as garbage.

2.3.3 Garbage Reclamation

Once the collector identifies the garbage objects there are several techniques that reclaim the space.

Direct to free list With reference counting, the collector may directly return the space used by garbage objects to a free list when an objects reference count falls to zero.

Sweep A sweep traverses over all the allocated objects in the heap, freeing the space associated with dead objects. For tracing garbage collection approaches that indirectly identify garbage, some form of sweep is required to free the space of the garbage objects. A sweep may operate over individual free list cells and larger regions of memory.

Compaction Compaction rearranges live objects within a region to create larger regions of free memory for future allocation. A classic example is sliding compaction where all live objects are compressed into a contiguous region of used memory, leaving a contiguous region of memory free for future allocation [Styger, 1967].

Evacuation Live objects can be evacuated from a region of memory and copied into another, to create an entire evacuated free region for future allocation. Evacuation requires two different regions of memory. Evacuation can be particularly effective when there are very few survivors since the cost is proportional to moving the survivors, and naturally aligns itself with tracing as the identification approach.

2.4 Canonical Garbage Collectors

The canonical garbage collectors use the above approaches for object allocation, garbage identification, and object reclamation.

2.4.1 Mark-Sweep

The first garbage collection algorithm was created as part of the LISP system [McCarthy, 1960], and is today known as mark-sweep. Mark-sweep is a tracing collector that runs in two simple phases. The *mark* phase performs a transitive closure over the object graph, marking each object as it is visited. The *sweep* phase performs a scan over all of the objects. If an object is not marked, it is unreachable and can be collected. On the other hand, if an object is marked, it is reachable, so the collector can clear the mark and retain it. It is possible for part of this sweeping phase to be performed by the mutator, a technique called *lazy sweeping* [Hughes, 1982]. Lazy sweeping reduces garbage collection pause time in stop the world collectors and can improve overall performance, due to improved cache behavior as sweep operations and subsequent allocations are performed on the same page in quick succession.

Mark-sweep is very efficient at collection time, the fastest and simplest full heap garbage collection mechanism available. This advantage is offset by slow downs it imposes on the mutator. Its free-list allocator forces the mutator to allocate objects in the discovered holes surrounding live objects. This policy often results in lower allocation performance, but the dominant effect is poor locality of reference due to objects being spread over the heap. Mark-sweep also suffers from fragmentation, where the total available memory may be sufficient to support an allocation request, but an empty, contiguous region of sufficient size may not be available.

2.4.2 Reference Counting

Reference counting was the second garbage collection algorithm published, also for the LISP system [Collins, 1960]. Reference counting operates under a fundamentally different strategy from that of tracing garbage collectors. Reference counting tracks for each object a count of the number of incoming references to it held by other objects, termed as the *reference count*. Whenever a reference is created or copied, the collector increments the reference count of the object it references, and whenever a reference is destroyed or overwritten, the collector decrements the reference count of the object it references. Write barriers implement counting. If an object's reference count reaches zero, the object has become inaccessible, and the collector reclaims the object and decrements the reference count of all objects referenced by that reclaimed object. Removing a single reference can potentially lead to many objects being freed. A common variation of the algorithm allows reference counting to be made incremental; instead of freeing an object as soon as its reference count becomes zero, it is added to a list of unreferenced objects, and periodically one or more items from this list are freed [Weizenbaum, 1969]. This naive algorithm is simple, inherently incremental, and requires no global computation. The simplicity of this naive implementation is particularly attractive and thus widely used, including in well-established systems such as PHP, Perl, and Python.

Reference counting has two clear limitations. It is unable to collect cycles of garbage because a cycle of references will self-sustain non-zero reference counts. Moreover, naive implementation of reference counting performs increment and decrement operations on every reference operation, including those to variables in stacks and registers. This overhead is further increased on multi-threaded systems, because the collector must perform reference count updates atomically to maintain correct counts. Prior optimizations overcome some of these limitations (see Section 2.5), and this thesis overcomes the remaining performance limitations.

2.4.3 Semi-Space

Semi-space collection is a tracing collector that uses evacuation to reclaim memory. The available heap memory is divided into two equal sized regions. During the execution of the program, the program allocates and uses objects in one region while the other region is empty. When garbage collection is triggered, the region containing objects is labeled as *from-space* and the empty region is labeled as *to-space*. The garbage collector performs a transitive closure over the object graph as in a mark-sweep collector. But instead of simply marking the object, each live object is copied when it is first encountered from the from-space to the to-space. It leaves a forwarding pointer to the new location in the old location. The collector then updates all references to the old location so that they point to the copied objects in the to-space. At the end of the collection, all reachable objects now reside in the to-space and the collector reclaims the whole from-space. At the start of each collection, roles of the spaces are switched. Initial implementations of semi-space collectors used a recursive algorithm [Minsky, 1963; Fenichel and Yochelson, 1969], which has unbound depth, but Cheney [1970] later implemented a simple iterative algorithm.

Semi-space collection makes less efficient use of memory because it must reserve half of the total memory to ensure there is space to copy all objects in the worst case. It can be expensive in collection time when all live objects need to be copied, but if very few objects survive it is efficient. It has good allocation performance and induces good mutator locality because contemporaneously allocated objects are allocated contiguously. During copying, the semi-space collector automatically compacts the heap, so it does not suffer from fragmentation.

2.4.4 Mark-Compact

Mark-compact collection aims to combine the benefits of both semi-space and mark-sweep collection. It addresses the fragmentation often seen in the mark-sweep collection by compacting objects into contiguous regions of memory. But it does so in place rather than relying on the large reserved space required by the semi-space collection. While this in-place transition saves space, it involves significant additional collection effort because it must traverse objects many times. The simplest form is sliding compaction, originally implemented in LISP-2 as a four phase algorithm [Styger, 1967]. In the first phase, the collector performs a transitive closure over the object graph, marking objects as they are visited. In the second phase, the collector calculates the future location of each marked object and remembers that location for each object. In the third phase, the collector updates all references to reflect the addresses calculated in the second phase. In the fourth and final phase, the collector copies objects to their new locations. Copying is done in strict address order to ensure that no live data is overwritten.

The additional phases of simple compaction algorithms make them significantly more expensive than simple mark-sweep or semi-space collection. While optimized versions of mark-compact collectors exist, they are rarely used as the primary collector in high performance systems. However, compaction is commonly combined with mark-sweep collection to provide a means to escape fragmentation issues, and is sometimes used alongside semi-space collection to allow execution to continue when memory is tight [Sansom, 1991]. Compaction does, however, have the advantage of excellent mutator locality because it preserves allocation order, and has very low space overheads.

2.4.5 Mark-Region

Mark-region collection combines contiguous allocation of semi-space with the collection strategy of mark-sweep. The motivation is to achieve the mutator performance of semi-space and the collection performance of mark-sweep. In terms of allocation, mark-region is similar to semi-space, with objects allocated into contiguous regions of memory. In terms of collection, mark-region is similar to mark-sweep, but instead of sweeping individual objects it sweeps regions; regions with no reachable objects are made available for allocation. A mark-region collection consists of two phases. The *mark* phase performs a transitive closure over the object graph, it marks objects as well their regions as they are visited. The *sweep* phase scans all the regions, and regions that were not marked in the mark phase are made available for future allocation. Immix provides the first detailed analysis and description of a mark-region collector [Blackburn and McKinley, 2008], although a mark-region approach was previously used in Oracle's JRockit and IBM's production virtual machines.

Mark-region collection combines excellent allocation performance and good collection performance, with good mutator locality. But it is susceptible to issues with fragmentation, because a single live object may keep an entire region alive and unavailable for reuse. To combat this problem, mark-region collectors often employ techniques to relocate objects in memory to reduce fragmentation. The JRockit collector performs compaction of the heap at each collection, the IBM collector performs whole heap compaction when necessary. Immix combats this problem by defining hierarchy of two regions (blocks divided into lines) and then adds lightweight defragmentation as required when memory is in demand. Immix is described further in Section 2.6.

2.4.6 Generational

Generational garbage collection is based on the weak generational hypothesis that 'most object dies young' [Ungar, 1984; Lieberman and Hewitt, 1983] and was at that time the most important advancement in garbage collection since the first collectors were developed in 1960. Generational collectors divide the heap into regions for objects of different ages, and perform more frequent collections on more recently allocated objects and less frequent collections on the oldest objects. The youngest generation is generally known as the *nursery* and a *minor* collection collects only the nursery. The space containing the oldest objects is known as the *mature* space. A full heap *major* collection collects both young and old generation. During a minor collection, generational collectors must remember all references from the mature space into the nursery and assume they are live. A *generational write barrier* takes note of references from older generations to younger generations, keeping them in a *remembered set* for use during minor collections. The remembered set in combination with the standard root set provide the starting point for a transitive closure across all live objects within the nursery. A partial copying collection can be performed during this closure, with live objects evacuated from the nursery into the mature space. When the weak generational hypothesis holds, this collection is very efficient because only a small fraction of nursery objects survive and must be copied into the mature space. Various mature space strategies are possible.

In generational collection, objects are allocated in the nursery contiguously, providing good allocation performance and better mutator locality. The collection performance is also very good compared to full heap collectors. The majority of high performance collectors (e.g., HotSpot, J9, and .NET) are generational collectors.

In the next section, we describe prior work on reference counting with different optimizations on which we build.

2.5 Reference Counting Garbage Collection

The first account of reference counting was published by George Collins in 1960, just months after John McCarthy described tracing garbage collection [McCarthy, 1960]. Reference counting directly identifies dead objects by keeping a count of the number of references to each object, freeing the object when its count reaches zero.

2.5.1 Naive Reference Counting

Collins' simple, *immediate* reference counters count *all* references, in the heap, stacks, registers and local variables. The compiler inserts increments and decrements on referents where ever references are created, copied, destroyed, or overwritten. Because such references are very frequently mutated, immediate reference counting has a high overhead. However, immediate reference counting needs very minimal runtime support, so is a popular implementation choice due to its low implementation burden. The algorithm requires just barriers on every pointer mutation and the capacity to identify all pointers within an object when the object dies. The former is easy to implement, for example compilers for statically and dynamically typed languages directly and easily identify pointer references, as do *smart pointers* in C++; while the latter can be implemented through a destructor. Objective-C, Perl, Delphi, PHP, and Swift [Stein, 2003; Thomas et al., 2013; Apple Inc., 2013, 2014] use naive reference counting.

Collins' reference counting algorithm suffers from significant drawbacks including: a) an inability to collect cycles of garbage, b) overheads due to tracking frequent stack pointer mutations, c) overheads due to storing the reference count, and d) overheads due to maintaining counts for short lived objects. We now briefly outline five important optimizations developed over the past fifty years to improve over Collins' original algorithm.

2.5.2 Deferral

To mitigate the high cost of maintaining counts for rapidly mutated references, Deutsch and Bobrow [1976] introduced deferred reference counting. Deferred reference counting ignores mutations to frequently modified variables, such as those stored in registers and on the stacks. Deferral requires a two phase approach, dividing execution into distinct mutation and collection phases. This tradeoff reduces reference counting work significantly, but delays reclamation. Since deferred references are not accounted for during the mutator phase, the collector counts other references and places zero count objects in a zero count table (ZCT), deferring their reclamation. Periodically in a GC reference counting phase, the collector enumerates all deferred references from the stacks and registers into a root set and then reclaims any object in the ZCT that is not a referent of the root set.

Bacon et al. [2001] eliminate the zero count table by buffering decrements between collections. Initially the buffered decrement set is empty. At collection time, the collector temporarily increments a reference count to each object in the root set and then processes all of the buffered decrements. Deferred reference counting performs all increments and decrements at collection time. At the end of the collection, it adds a buffered decrement for every root. Although much faster than naive immediate reference counting, deferred reference counting typically uses stack maps [2.7.1] to enumerate all live pointers from the stacks. Stack maps are an engineering impediment, which discourages many reference counting implementations from including deferral [Jibaja et al., 2011].

2.5.3 Coalescing

Levanoni and Petrank [2001, 2006] observed that all but the first and last in any chain of mutations to a reference within a given window can be *coalesced*. Only the initial and final states of the reference are necessary to calculate correct reference counts. Intervening mutations generate increments and decrements that cancel each other out. This observation is exploited by remembering (logging) only the initial value of a reference field when the program mutates it between periodic reference counting collections. At each collection, the collector need only apply a decrement to the initial value of any overwritten reference (the value that was logged), and an increment to the latest value of the reference (the current value of the reference).

Levanoni and Petrank implemented coalescing using *object remembering*. The first time the program mutates an object reference after a collection phase: a) a write barrier logs *all* of the outgoing references of the mutated object and marks the object as logged; b) all subsequent reference mutations in this mutator phase to the (now logged) object are ignored; and c) during the next collection, the collector scans the remembered object, increments *all* of its outgoing pointers, decrements all of its remembered outgoing references, and clears the logged flag. This optimization uses two buffers called the mod-buf and dec-buf. The allocator logs all new objects, ensuring that outgoing references are incremented at the next collection. The allocator does *not* record old values for new objects because all outgoing references start as *null*.

2.5.4 Ulterior

Blackburn and McKinley [2003] introduced *ulterior reference counting*, a hybrid collector that combines copying generational collection for the young objects and reference counting for the old objects. They observe that most mutations are to the nursery objects. Ulterior reference counting extends deferral to include all nursery objects. It restricts copying to nursery object and reference counting to old objects, the object demographics for which they perform well. It safely ignores mutations to select heap objects. Ulterior reference counting is not difficult to implement, but the implementation is a hybrid, and thus manifests the implementation complexities of both a standard copying nursery and a reference counted heap. Azatchi and Petrank [2003] independently proposed the use of reference counting for the old generation and tracing for the young generation. During a nursery collection, their collector marks all live nursery objects, and sweeps the rest. Their reclamation is thus proportional to the entire nursery size, rather than the survivors as in a copying nursery. They also explored the use of the reference counting buffers of update coalescing as a source of inter-generational pointers. This collector is also hybrid, so manifests the implementation complexities of two orthodox collectors.

2.5.5 Age-Oriented

Paz et al. [2005] introduced *age oriented* collection, which aimed to exploit the generational hypothesis that most objects die young. Their age-oriented collector uses a reference counting collection for the old generation and a tracing collection for the young generation that establishes reference counts during tracing. This collector provides a significant benefit as it avoids performing expensive reference counting operations for the many young objects that die. It does not perform copying. Like ulterior reference counting, this collector is a hybrid, so manifests the implementation complexities of two orthodox collectors.

2.5.6 Collecting Cyclic Objects

Reference counting suffers from the problem that cycles of objects will sustain non-zero reference counts, and therefore cannot be collected. There exist two general approaches to deal with cyclic garbage: *backup tracing* [Weizenbaum, 1969; Frampton, 2010] and *trial deletion* [Christopher, 1984; Martinez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Paz et al., 2007]. Paz et al. [2007] compared backup tracing with trial deletion and found that backup tracing performed slightly better and predicted in future trial deletion will outperform backup tracing. Later Frampton [2010] conducted a detailed study of cycle collection and showed that backup tracing, starting from the roots, performs significantly better than trial deletion and has more predictable performance characteristics.

2.5.6.1 Backup Tracing

Backup tracing performs a mark-sweep style trace of the entire heap to eliminate cyclic garbage. The only key difference to a classical mark-sweep is that during the sweep phase, decrements must be performed from objects found to be garbage for their descendants into the live part of the heap. To support backup tracing, each object needs to be able to store a mark state during tracing. Backup tracing can also be used to restore *stuck* reference counts due to use of limited bits to store the count. Backup tracing must enumerate live root references from the stacks and registers, which requires stack maps. For this reason, naive reference counting implementations usually do not perform backup tracing.

2.5.6.2 Trial Deletion

Trial deletion collects cycles by identifying groups of self-sustaining objects using a partial trace of the heap in three phases. In the first phase, the sub-graph rooted from a selected candidate object is traversed, with reference counts for all outgoing pointers (temporarily) decremented. Once this process is complete, reference counts reflect only external references into the sub-graph. If any object's reference count is zero then that object is only reachable from within the sub-graph. In the second phase, the sub-graph is traversed again, and outgoing references are incremented from each object whose reference count did not drop to zero. Finally, the third phase traverses the sub-graph again, sweeping all objects that still have a reference count of zero. The original implementation was due to Christopher [1984] and has been optimized over time [Martinez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Paz et al., 2007].

Lins [1992] performed cycle detection lazily, periodically targeting the set of candidate objects whose counts experienced decrements to non-zero values. Bacon and Rajan [2001] performed each phase over all candidates in a group after observing that performing the three phases for each candidate sequentially like Lins could exhibit quadratic complexity. Bacon and Rajan also used simple static analysis to exclude the processing of objects they could identify as inherently acyclic. Paz et al. [2007] combines the work of Bacon and Rajan [2001] with update coalescing that significantly reduces the set of candidate objects. They also develop new filtering techniques to filter out a large fraction of the candidate objects. They also integrate their cycle collector with the age oriented collector [2.5.5] that greatly reduces the load on the cycle collector.

2.5.7 Reference Counting and the Free List

Free lists support immediate and fast reclamation of individual objects, which makes them particularly suitable for reference counting. Other systems, such as evacuation and compaction, must identify and move live objects before they may reclaim any memory. Also, free lists are a good fit to backup tracing used by many reference counters. Free lists are easy to *sweep* because they encode free and occupied memory in separate metadata. The sweep identifies and retains live objects and returns memory occupied by dead objects to the free list.

Blackburn et al. [2004a] show that contiguous allocation in a copying collector delivers significantly better locality than free-list allocation in a mark-sweep collector. When contiguous allocation is coupled with copying collection, the collector must update all references to each moved object [Cheney, 1970], a requirement that is at odds with reference counting's local scope of operation. Because reference counting does not perform a closure over the live objects, in general, a reference counting collector does not know of and therefore cannot update all pointers to an object it might otherwise move. Thus far, this prevented reference counting from copying and using a contiguous allocator.

This section provided a detailed overview on how reference counting works, its shortcomings, and different optimizations to overcome some of them. This section also explained why reference counting uses a free list for allocation. These are the necessary background for Chapter 4. In the next section, we describe Immix garbage collection.

2.6 Immix Garbage Collection

As introduced in Section 2.4.5, mark-region memory managers use a simple bump pointer to allocate objects into regions of contiguous memory [Blackburn and McKinley, 2008]. A tracing collection marks each object and marks its containing region. Once all live objects have been traced, it reclaims unmarked regions. This design addresses the locality problem in free-list allocators. A mark-region memory manager can choose whether to move surviving objects or not. By contrast, evacuating and compacting collectors must copy, leading them to have expensive space (semi-space) or time (mark-compact) collection overheads compared to mark-sweep collectors. Mark-region collectors are vulnerable to fragmentation because a single live object may keep an entire region alive and unavailable for reuse, and thus must copy some objects to attain space efficiency and thus performance.

2.6.1 Heap Organization: Lines and Blocks

Immix is a mark-region collector that uses a region hierarchy with two sizes: lines, which target cache line locality, and blocks, which target page level locality [Blackburn and McKinley, 2008]. Each block is composed of lines, as shown in Figure 2.1. The bump pointer allocator places new objects contiguously into empty lines and skips over occupied lines. Objects may span lines, but not blocks. Several small objects typically occupy a line. Immix uses a bit in the header to indicate whether an object straddles lines, for efficient line marking. Immix recycles free lines in partially free blocks, allocating into them first. Figure 2.1 shows how Immix allocates objects contiguously in empty lines and blocks.

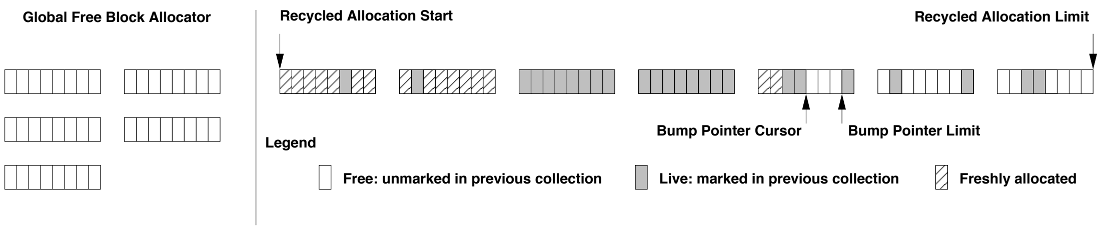
Global Free Block Allocator
Recycled Allocation Start
Recycled Allocation Limit
Bump Pointer Cursor
Bump Pointer Limit
Legend
Free: unmarked in previous collection
Live: marked in previous collection
Freshly allocated

Figure 2.1: Immix Heap Organization [Blackburn and McKinley, 2008]

2.6.2 Opportunistic Copying

Immix tackles fragmentation using *opportunistic* defragmentation, which mixes marking with copying. At the beginning of a collection, Immix determines whether to defragment, e.g., based on fragmentation levels. Blocks with free lines at the start of a collection indicate fragmentation because although available, the memory was not usable by the mutator. Furthermore, the live/free status for these blocks is up-to-date from the prior collection. During a defragmenting collection, Immix uses two bits in the object header to differentiate between marked and forwarded objects. At the beginning of a defragmenting collection, Immix identifies source and target blocks based on the statistics from the previous collection. During the mark trace, when Immix first encounters an object that resides on a source block and there is still available memory for it on a target block, Immix copies the object to a target block, leaving a forwarding pointer. Otherwise Immix simply marks the object as usual. When Immix encounters forwarded objects while tracing, it updates the reference accordingly. This process is opportunistic, since it performs copying until it exhausts free memory to defragment the heap. The result is a collector that combines the locality of a copying collector and the collection efficiency of a mark-sweep collector with resilience to fragmentation. The best performing production collector in Jikes RVM is generational Immix (Gen Immix), which consists of a copying nursery space and an Immix old space [Blackburn and McKinley, 2008].

This section provided a detailed overview of the Immix garbage collection. It described the line and block heap organization of Immix with contiguous allocation and the opportunistic copying mechanism to mitigate fragmentation. These are the necessary background for Chapter 5. In the next section, we describe conservative garbage collection.

~~2.7 Conservative Garbage Collection~~

~~Conservative collectors have thus far been tracing. A tracing garbage collector performs a transitive closure over the object reachability graph, starting with the *roots*, which are references into the heap held by the runtime, including stacks, registers, and static (global) variables. An *exact* garbage collector precisely identifies root ref-~~